Disks

Files

Blocks and clusters

Disks can be seen by an operating system as a collection of **data blocks** that can be read or written independently.

To allow for disk management, each block is characterised by a unique numerical address called **Logical Block Address** (or *LBA*).

Typically, the operating system groups blocks into **clusters**, the minimal unit an operating system can read from or write to a disk. Typical cluster sizes range from one disk sector (512 B) to 128 sectors (64 kB).

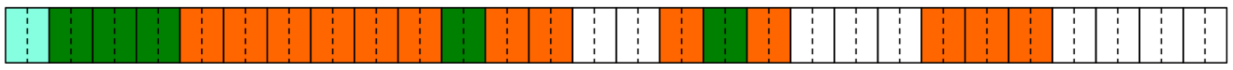
Clusters contain two things:

- **File data**: the content of the files saved on the disk
- **Metadata**: the information about the files required by the file system

Metadata usually contains:

- File names
- Directory structures
- Symbolic links
- File size
- File type
- Creation, modification and last access dates
- Security information
- Links to the LBA where the content can be located on the disk

The disk thus contains several types of cluster:



🟦 Meta data – fixed position (to bootstrap the entire file system)

🟩 Meta data – variable position (to hold the folder structure)

🟧 File data (actual content of the files)

⬜ Unused space (available to contain new files and folders)

Reading and writing files

Reading a file requires:

1. Accessing the metadata to locate the blocks that store that file
2. Accessing the blocks to read such file

Writing a file, instead, requires:

1. Accessing the metadata to locate free blocks on the disk
2. Writing the file data to those blocks

Deleting files

In order to delete a file from a disk, there is no need to actually delete the data on the logical blocks of the disk: it is sufficient to update the metadata on the disk and marking the blocks containing the deleted files as unused space.

Internal fragmentation

Since the file system can only access clusters, the real occupation of space on a disk for a file is always a multiple of the cluster size. If we call s the file size, c the cluster size and a the actual size of a file on the disk, we can calculate:

$$a = \left\lceil \frac{s}{c} \right\rceil \cdot c$$

Knowing this, we can say that the quantity:

$$w = a - s$$

is **wasted disk space**. This waste is called **internal fragmentation** of files.

External fragmentation

As the life of the disk evolves, there might not be enough space to store a file in contiguous block. In this case, the file is split into smaller chunks which are inserted into the free clusters spread over the disk.

Splitting a file into non-contiguous clusters on a disk is called **external fragmentation**. This can reduce performance on an HDD by a long margin.

Hard Disk Drives (HDDs)

General information

A **Hard Disk Drive** (or *HDD*) is a data storage unit which uses magnetic disks, called *platters*, coated with magnetic material to store data.

Data is read in a random-access manner: individual blocks can be stored or retrieved in any order.

Externally, HDDs expose a large number of sectors, typically 512-4096 bytes in size, which have a header, error correcting code and in which writes are atomic. Only multiple sector writes can be interrupted: this is called a *torn write*.

On HDDs, sectors are arranged into tracks and a cylinder is a particular track on multiple platters. Tracks are arranged in concentric circles on platters. A disk may have multiple platters which, in turn, are double-sided. Modern HDDs usually have a track density of 16000 Tracks per Inch (TPI).

HDDs have drive motors which spin the platters at a constant rate, which is measured in *Revolutions Per Minute* (RPM). Usual speeds on modern HDDs range from 7200 RPM to 15000 RPM.

Caching

Many disks also incorporate caches, which act as *track buffers*. Caches are usually pretty small on HDDs, ranging from 8 to 32 MB of storage.

Caches can be used to perform both read caching and write caching: read caching reduces read delays caused by seeking and rotation; write caching is used to speed up writes to the disk.

Two types of write caches are used in HDDs:

- **Write-back** cache: writes are reported as complete after they have been cached; this can be dangerous especially if the HDD goes offline before the write was moved from cache to disk.
- **Write-through** cache: writes are reported as complete after they have been written to the cache and, subsequently, to the disk.

Some disks include flash memory instead of RAM for persistent caching: these are a type of disk called Solid State Hybrid Drives (or SSHD).

Delays

In a HDD, there are four types of delay that can be measured with every I/O operation:

1. **Rotational** delay: the time needed for the disk to rotate to the desired sector, so that it can be read by the read head:

$$T_{\text{rotation}} = \frac{60\ s/min}{2 \cdot R_{\text{RPM}}}$$

2. **Seek** delay: the time needed to move the read head to the wanted track:

$$T_{\text{seek}} = \frac{T_{\text{seek}}^{\text{max}}}{3}$$

3. **Transfer** time: the time needed to transfer data from the disk elsewhere in the system:

$$T_{\text{transfer}} = \frac{\text{Size of block to transfer}}{\text{Transfer speed}}$$

4. **Controller** overhead: the overhead for the request management

The **service time** for an HDD is the sum:

$$T_{I/O} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}} + T_{\text{overhead}}$$

We can also define a hard drive's **response time** as:

$$\tilde{R} = T_{\text{queue}} + T_{\text{I/O}}$$

where T_{queue} depends on queue length, resource utilisation, mean and variance of disk service time and request arrival distribution.

The service time considered above, however, considers the very pessimistic case in which sectors are fragmented in the worst possible way on the disk. In many circumstances, this is not the case. We can remedy this by measuring the **data locality** of a disk as the percentage of blocks that do not need seek or rotational latency to be found. We can thus redefine our service time as:

$$T'_{I/O} = (1 - \text{DL}) \cdot (T_{\text{seek}} + T_{\text{rotation}}) + T_{\text{transfer}} + T_{\text{overhead}}$$

Disk scheduling

While caching helps improve poor disk performance, it can't make up for poor random access times. However, if there is a queue full of requests ready to be served by the disk, they could be reordered so to optimise access times.

We will consider the following scheduling algorithms:

- **First Come First Serve**: the most basic algorithm for disk scheduling: the requests sent to the disk are simply served in the order they arrive in
- **Shortest Seek Time First**: improves latency by only moving to the request "nearest" to the currently served one first
- **SCAN**: the disk maintains a sweeping motion when serving requests
- **Circular SCAN**: same as SCAN, but the sweeping motion is made in only one direction
- **Circular LOOK**: a more optimised C-SCAN, which takes into consideration the lowest and highest requests

First Come First Serve (FCFS)

This is the most basic algorithm for disk scheduling. Once the requests reach the disk queue, they are served in order from first to last. This makes access times longer because of the distances the read arm has to travel to get requests that may be several cylinders apart.

Shortest Seek Time First (SSTF)

This algorithm improves on FCFS by significantly decreasing the distance between served requests. In fact, they are ordered by shortest seek time in order to optimise the read arm's movement.

For example, with the following list of requests:

$$98 \to 183 \to 37 \to 122 \to 14 \to 124 \to 65 \to 67$$

assuming the read head starts from block 53, we get the following request order:

$$65 \to 67 \to 37 \to 14 \to 98 \to 122 \to 124 \to 183$$

The problem with this approach is that it is prone to starvation: if more requests come, the requests that have to reach farthest in the disk will always be delayed to make room for nearer ones.

SCAN

In this algorithm, the head "sweeps" across the disk, going to the lowest block addresses to the highest (or viceversa) and, once the arm reaches for the bounds of the disk, the read arm flips its direction and returns back, serving the leftover requests.

Using the same request list as before:

$$98 \to 183 \to 37 \to 122 \to 14 \to 124 \to 65 \to 67$$

and assuming the read head goes to the lowest addresses first starting at block 53, the reordered requests are:

$$53 \to 37 \to 14 \to [\text{flip}] \to 65 \to 67 \to 98 \to 122 \to 124 \to 183$$

This makes for a starvation-free algorithm at the cost of worsening performance for requests that are at very low or very high addresses.

Circular SCAN (C-SCAN)

This algorithm follows the same principle of SCAN, with a catch: the read head only follows one direction.

If we take our usual request list:

$$98 \to 183 \to 37 \to 122 \to 14 \to 124 \to 65 \to 67$$

and assume the read head moves from low to high addresses starting from block 53, we obtain:

$$65 \to 67 \to 98 \to 122 \to 124 \to 183 \to \text{[restart from bottom]} \to 14 \to 37$$

This renders the C-SCAN algorithm fairer than SCAN, but with a worse performance.

Circular LOOK (C-LOOK)

This is but a variant of the C-SCAN algorithm which peeks at the upcoming values of the queue in order to “wrap around” the disk as soon as there are no more requests ahead.

Solid State Drives (SSDs)

Solid State Drives are an evolution of Hard Disk Drives which use no mechanical or moving parts but instead save data through transistors. This makes for a higher performance disk with the same interfaces as a HDD.

Storing bits

In order to save data, a SSD applies voltage to a Floating Gate MOSFET (FGMOS) to encode one or more bits to it. Depending on how many bits are encoded to a single cell in the disk’s flash memory, we can have:

- Single-level cell (SLC): one bit per cell
- Multi-level cell (MLC): two bits per cell
- Triple-level cell (TLC): three bits per cell
- Quad-level cell (QLC): four bits per cell
- Penta-level cell (PLC): five bits per cell, currently in development

SLC encoding is the most reliable, but also the least cost-effective: it costs more per GB than other encoding methods.

Internal organisation

NAND flash is organised into **pages** and **blocks**: a page contains multiple logical block addreses, while a block consists of multiple pages. These two units have specific purposes: blocks, also called *erase blocks*, are the smallest unit on SSDs that can be erased; pages, on the other hand, are the smallest unit that can be read or written to.

Pages can have three states in a SSD:

- **Dirty** or **Invalid**: the page contains data that is no longer in use by the file system
- **Empty** or **Erased**: the page does not contain data
- **In use** or **Valid**: the page contains data that is still used by the file system

Writing and erasing

The “rules” for writing and erasing on a SSD are:

- Only empty pages can be written
- Only dirty blocks—blocks that only contain dirty pages—can be erased
- It is only meaningful to read pages in the valid state
- If no empty page exists, some dirty page must be erased to write new data

As should be clear from these rules, if no dirty block exists, one must be made by rearranging dirty pages into one block, moving other valid or empty pages around the disk. Furthermore, in order to erase a block in a SSD, the original voltage must be reset to neutral, which brings forth the problem of **write amplification**: the actual amount of information that is physically written to the storage media is a multiple of the logical amount intended to be written.

Write amplification

Let’s assume we buy a new SSD with a page size of 4 kB, a block size of 5 pages and a drive size of 1 block. Let’s also assume that the read speed of such SSD is 2 kB/s and its write speed 1 kB/s.

Let’s now write a 4 kB text file on the disk: it takes four seconds. If we continue by writing an 8 kB picture to it, this operation takes eight seconds, as expected.

Now, let’s delete our text file: this leaves us with two “valid” pages, one “dirty” page and two “empty” pages.

Finally, let’s write another picture to the disk, this time a 12 kB one. Theoretically, this should take 12 seconds; however it takes double the amount. This happens because of write amplification: according to the rules we listed earlier, to write the 12 kB picture, the disk has to:

1. Read its block into the cache
2. Delete the dirty page from the cache
3. Write the new picture into the cache
4. Delete the old block on the SSD
5. Write the cache back to the SSD

This means that while the operating system thought it would be writing 12 kB of data to the disk, it actually read 8 kB—the valid pages from the SSD to the cache—and then wrote 20 kB—the old valid pages and the new pages for the 12 kB picture from the cache to the disk—for a combined I/O time of 24 seconds.

With time, write amplification takes its toll on the performance of the SSD in a significant way.

Wearing of flash cells

Each time a block is erased, the disk sends a relatively high charge of electricity to the MOSFETs that contain that block. This degrades the silicon material of the MOSFETs themselves, which means that after enough write-erase cycles, the cells become unreliable and cannot store information correctly anymore.

Flash Translation Layer (FTL)

Direct mapping between logical and physical pages is not feasible, so a component called *Flash Translation Layer* (or FTL) is used to make an SSD "look like" an HDD to the operating system.

The FTL manages:

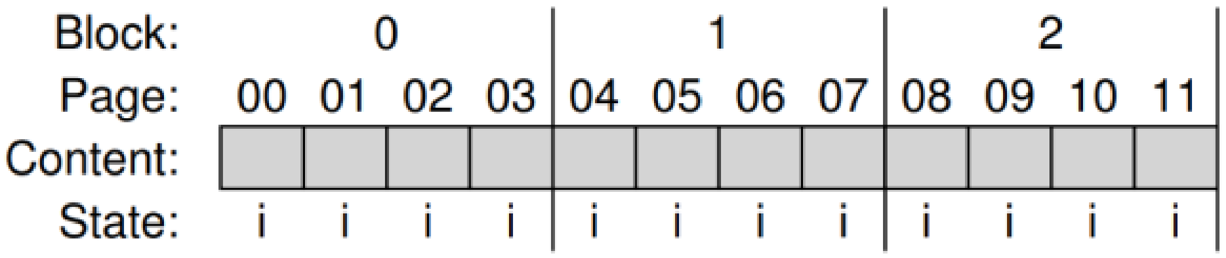
- **Data allocation** and **address translation**: it oversees write and erase processes in order to reduce write amplification effects
- **Garbage collection**: it reuses pages in the dirty state to make room for new data
- **Wear levelling**: writes are spread across blocks in the flash memory, which makes all blocks in the device wear out at roughly the same time

Log-structured FTL

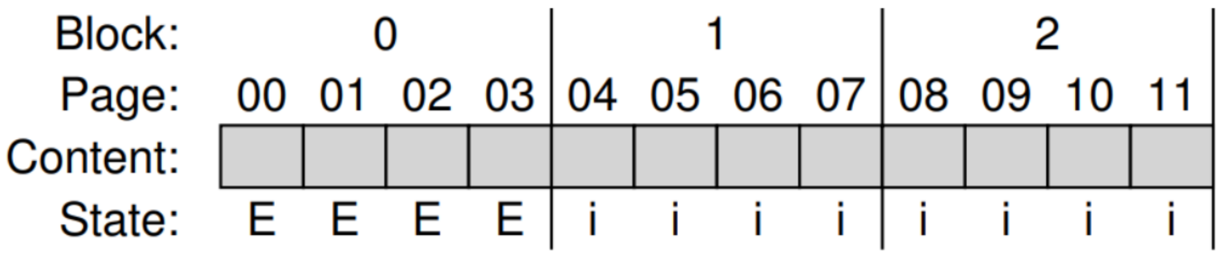
One way the FTL can work is by appending every write and erase operation to a log and optimising operations in order to minimise write amplification. For example, let's take the following log:

```
write(100, a1)
write(101, a2)
write(2000, b1)
write(2001, b2)
```

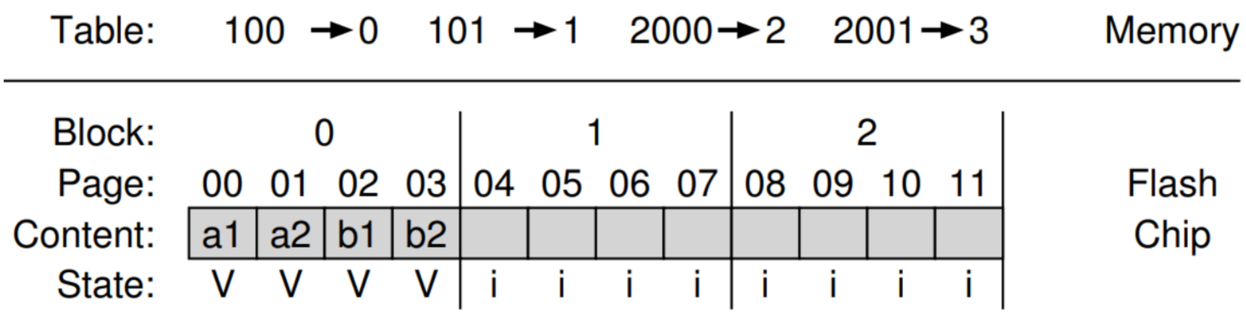
We start from an initial state in which all pages are marked as invalid:



If we assume that LBA 100 is mapped to block 0, page 00, we can then erase block 0 to make room for the data specified in the log:



We can then write our data in the newly created empty pages and set them as valid:



We also keep a table which maps logical blocks to physical blocks. Now, if we assume these two operations are appended to the log:

```
write(100, c1)
write(101, c2)
```

Instead of erasing block 0 again, which would cause write amplification, we just **change the mapping table** and **write the new data to a freshly erased block**:

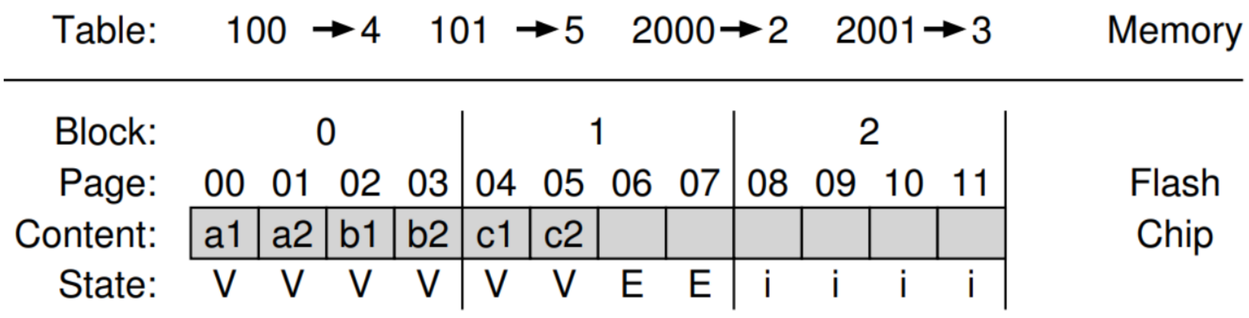


Table: 100 →4 101 →5 2000→2 2001→3 Memory

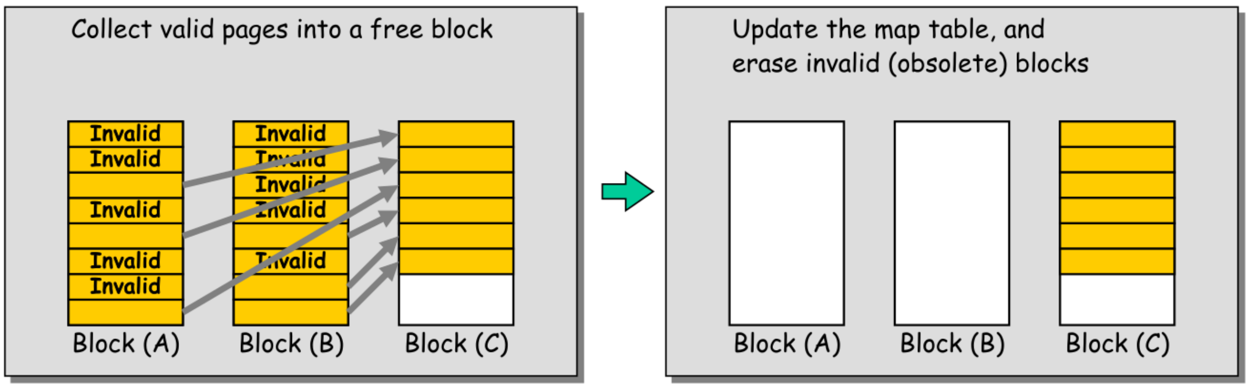
Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash Chip
Content:	a1	a2	b1	b2	c1	c2							
State:	V	V	V	V	V	V	E	E	i	i	i	i	

Garbage collection

When an existing page is updated, old data becomes obsolete. The old data that is no longer used is called garbage and, eventually, garbage must be collected in order to make room for new data.

Garbage collection is the process of finding garbage blocks and reclaiming them. This is very simple when a block only contains garbage pages, but becomes more complex when it contains both valid and garbage pages.

When a block contains both valid and invalid pages, in order to make room for new data, the garbage collector copies valid pages to an empty block, leaving only the invalid pages in the old, full blocks, and then erases them.



Garbage collection is expensive: it requires a lot of reading and writing of live data and its cost depends on the amount of data blocks that must be migrated to make free space. There are, however, solutions to alleviate the problem, like:

- Overprovisioning the device by adding extra flash capacity
- Running the garbage collector in the background when the disk is less busy

Since a file is never deleted from the disk when it is deleted from the operating system, the garbage collector tries to assume which pages are garbage and can be erased, but this doesn't always work: let's assume that a block contains two pages of metadata and all the remaining pages constitute a file. If the OS deletes that file, it will know that only the file pages are to be erased, while the metadata should remain. However, that is not the case for the garbage collector, which just assumes all pages are still valid.

This problem is solved by a new SATA command, called `TRIM`, which lets the OS tell the disk that specific LBAs are invalid and may be garbage collected.

Mapping table

As we said previously, a table is needed to translate from logical addresses to physical addresses on SSDs. This brings forth new problems, such as defining how big the mapping table should be.

For example, page-level mapping is inefficient: in a 1 TB SSD with a 4 byte entry per 4 kB page, 1 GB is needed for mapping. For this reason, some approaches try to reduce the costs of mapping, like:

- Block-based mapping: it maps at a coarser level, using less space overall
- Hybrid mapping: it uses multiple tables to make mapping more efficient
- Page mapping with caching: it exploits data locality to improve mapping size

Block mapping

In order to reduce the mapping table size, it is possible to map at block granularity. This makes the mapping table significantly smaller, but introduces the **small write problem**: let's assume a write log such as:

```
write(2000, a)
write(2001, b)
write(2002, c)
write(2003, d)
```

This would simply write a block like so:

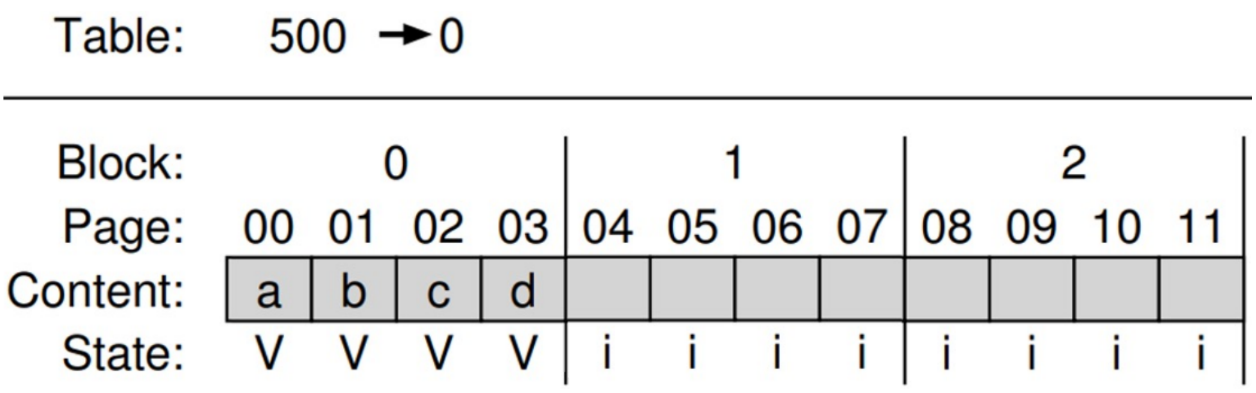


Table: 500 → 0
Block: 0 1 2
Page: 00 01 02 03 04 05 06 07 08 09 10 11
Content: a b c d
State: V V V V i i i i i i i i

However, if another very small write is appended to the log:

```
write(2002, c)
```

Then the entire block has to be moved:

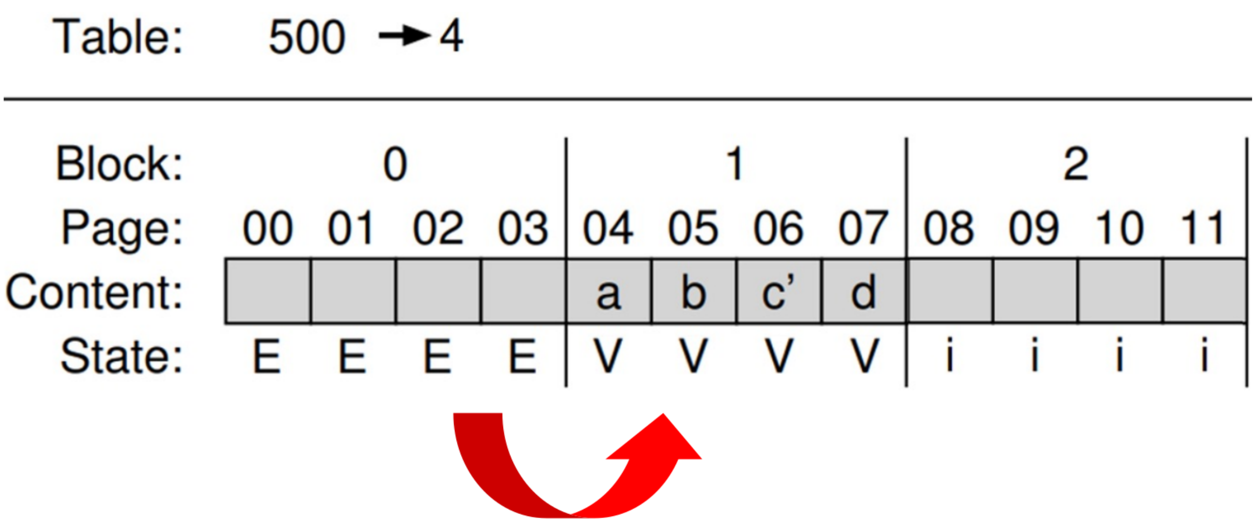


Table: 500 → 4
Block: 0 1 2
Page: 00 01 02 03 04 05 06 07 08 09 10 11
Content: a b c' d
State: E E E E V V V V i i i i

Hybrid mapping

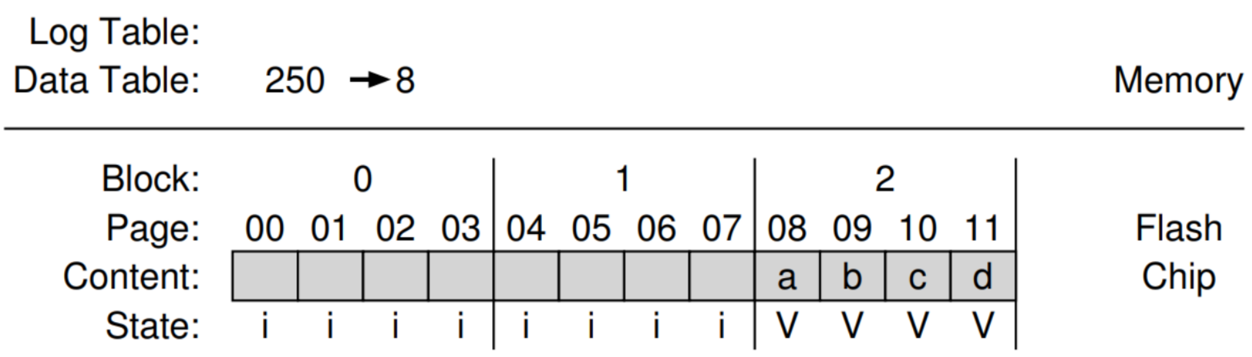
With hybrid mapping, the FTL maintains two tables:

- Log blocks, which are page-mapped
- Data blocks, which are block-mapped

When looking for a specific logical block, the FTL will consult the page-mapping table and block-mapping table in order. For example, let's assume the following writes:

```
write(1000, a)
write(1001, b)
write(1002, c)
write(1003, d)
```

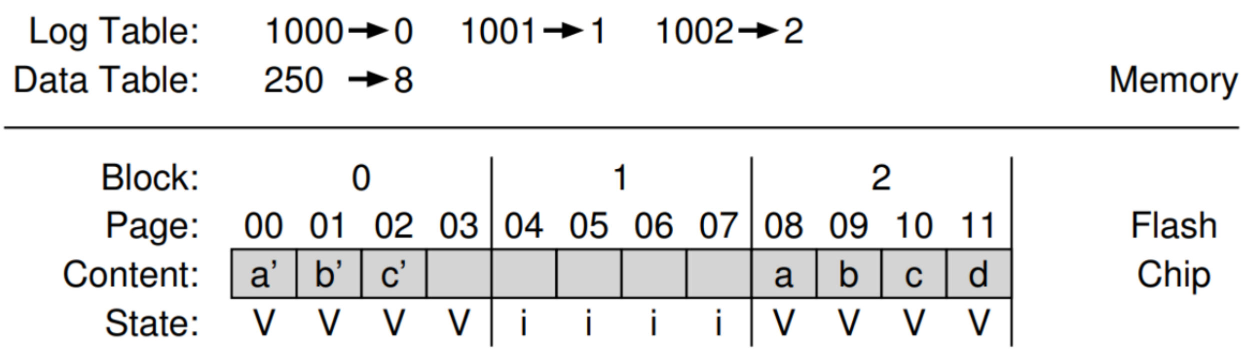
This will result in the following tables:



Now, if we have to update some of the previously written pages:

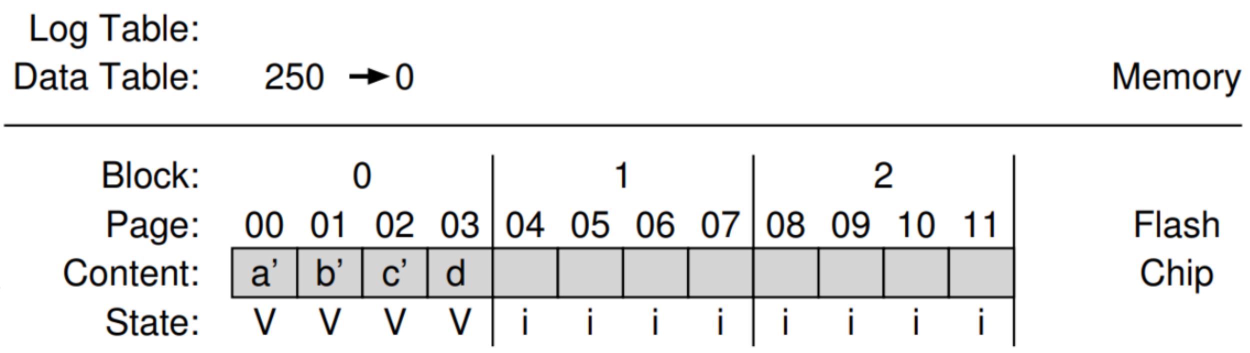
```
write(1000, a')
write(1001, b')
write(1002, c')
```

The FTL will simply write them somewhere else and update the log table:



Log Table: 1000 → 0 1001 → 1 1002 → 2
Data Table: 250 → 8
Memory
Block: 0 1 2
Page: 00 01 02 03 04 05 06 07 08 09 10 11
Content: a’ b’ c’ a b c d
State: V V V V i i i i V V V V
Flash Chip

When needed, the FTL can perform a `MERGE` operation in order to write the data “as it should be” on disk:



Page mapping with caching

The basic idea behind page mapping with caching is to cache the active part of the page-mapped FTL: if a given workload only accesses a small set of pages, the translations of those pages will be stored in the FTL memory.

This leads to high performance at a low memory cost if the cache can contain the necessary working set. However, cache miss overhead can be an issue.

Wear levelling

When using flash memory, erase/write cycles are limited. Furthermore, if an SSD is not evenly worn out, its lifespan is shortened.

A log-structured FTL approach combined with garbage collection helps in spreading writes, but a block which consists of cold data—data that is not overwritten often—should be periodically moved to another block in order to even-out wearing on the disk.

However, wear levelling increases the write amplification of the disk and decreases performance, so in order to avoid overdoing it, a simple policy can be applied by giving each flash block its own erase/write cycle counter:

$$\left|\mathrm{max}(\mathrm{e/w\ cycle}) - \mathrm{min}(\mathrm{e/w\ cycle})\right| < e$$